

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of:

Jeffrey A. BEDELL, et al.

Serial No.: 09/883,475

Filed: June 20, 2001

Art Unit: 2134

Examiner: N. Wright

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

DECLARATION OF PRIOR INVENTION UNDER 37 C.F.R. § 1.131

Sir:

We, Jeffrey A. Bedell, Benjamin Z. Li, Luis Orozco and Ramparasad Polana, hereby declare that we are co-inventors of the invention that is claimed in the above-identified patent application. Prior to January 4, 2001, we conceived of and reduced to practice the invention that is claimed in the above-identified patent application.

Internal documentation related to the development of this feature demonstrates conception as early as April 30, 1998 with diligence through its actual reduction to practice on or after June 20, 2000. Exhibit A is a May 25, 1998 document entitled "DSS Server Job Priority" that describes one embodiment of assigning priorities and servicing them as claimed. This feature was part of a major new product development at MicroStrategy given the internal code name Castor. Castor substantially rewrote the platform of the business intelligence software system of MicroStrategy. As a result, this feature, and many others, went through numerous rounds of tests, a confidential beta release and then final release on or before June of 2000 in a product called MicroStrategy 7.0.

We further hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that


these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

DECLARANT: 
Jeffrey A. Bedell

Date: 8-15-05

DECLARANT: 
Benjamin Z. Li

Date: 08-15-2005

DECLARANT: 
Luis Orozco

Date: 8/15/05

DECLARANT: 
Ramparasad Polana

Date: 8-17-05



DSS Server Job Priority

ROLE OF JOB PRIORITY IN JOB EXECUTION	3
1.1 ENTERING JOBS INTO A QUEUE.....	3
1.2 CALCULATION OF JOB PRIORITY	4
IMPLEMENTATION OF JOB PRIORITY	5
1.3 MSIJOBPRIORITYSCHEMEOBJECT	5
1.4 CREATION OF MSIJOBPRIORITYSCHEMEOBJECTS.....	6
1.5 PROPOSED IMPLEMENTATION ACCORDING TO SPECIFICATIONS	6
1.6 CHANGING PRIORITY AFTER JOB IS ENTERED INTO QUEUE	8
JOB SERVICING SCHEMES.....	9
1.7 UNITS OF INDEPENDENT RESOURCE ALLOCATION AND CONTROL.....	9
1.8 JOB PROCESSING ACCORDING TO A SERVICING SCHEME	10
1.8.1 <i>FixedThreadCooperative</i>	11
1.8.2 <i>WeightedShare</i>	11
1.8.3 <i>HighestPriorityFirst</i>	12

ABSTRACT

This document describes the job priority computation and usage in DSS Server which affects the execution order of user requests.

HISTORY

Date	Author	Description
4/30/98	Ramprasad Polana	Initial Version
5/13/98	Ramprasad Polana	Added servicing schemes
5/25/98	Ramprasad Polana	Added changes in job priority schemes as decided through internal team review and CTA reviews

REFERENCES:

A must read: Castor server specification document: section on job prioritization and servicing.

ROLE OF JOB PRIORITY IN JOB EXECUTION

DSS Server creates job objects for every user requests that can not be immediately serviced. Processing units within DSS Server execute jobs in a pipeline architecture (refer to the server internal architecture document for details on the pipeline architecture). Each processing unit contains a hierarchical queue (we will call this a station) and a pool of threads that service the queues. The jobs are placed within a queue based on its priority while the threads available to service a job will pick the first job within a queue that is selected using the servicing schemes.

Following two sections describe the process of entering the job into a particular queue of a processing unit.

1.1 ENTERING JOBS INTO A QUEUE

Every processing unit in DSS Server contains a hierarchical queue, which is organized as a tree. Typical processing units contain only a two-level hierarchy (except for the processing units containing the DSSQueryExecutionTask which require three-level hierarchy where the first level split by the warehouse dbc type). The leaf nodes represent basic FIFO queues, while all other nodes are a collections of queues, or queue sets.

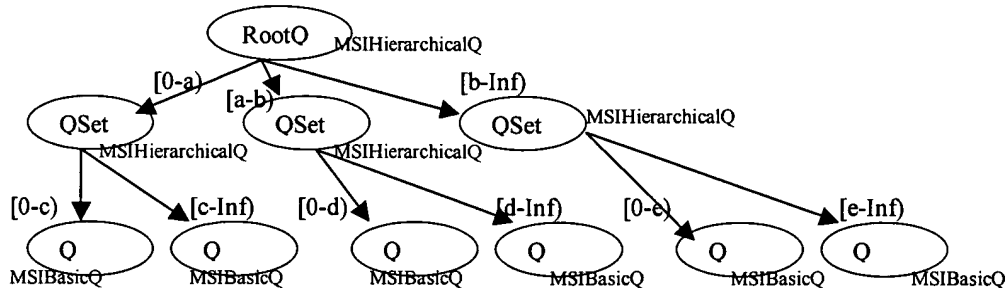


Figure 1. Hierarchical queue structure in a typical DSS Server PU. The variables a, b, c, d and e are positive integers and Inf stands for infinity.

```

MSIQRtStatus MSIHierarchicalQ::Enqueue(JOB_TYPE *iJob, unsigned long iMilliseconds =
gcInfiniteTimeout){
    int aPriority = mPrioritySchemeObject->CalculatePriority(iJob);
    MSIQRtStatus aRC;
    for(int aSubQ = 0; aSubQ < mSubQueues; aSubQ++){
        if(aPriority < mUpperBounds[aSubQ]){
            aRC = mSubQ[aSubQ] -> Enqueue(iJob, iMilliseconds);
            break;
        }
    }
    if(aSubQ >= mSubQueues) {
        aRC = mSubQ[mSubQueues-1] -> Enqueue(iJob, iMilliseconds);
    }
    return aRC;
}

```

When a job needs a particular task to be done, the JobExecutor inside DSS Server finds a processing unit which can perform that task, and hands over the job to the processing unit by way of the MSIPU::Enter(MSIJob *iJob) method. The implementation of this method in turn, calls the MSIHierarchicalQ::Enter(MSIJob *iJob) method to place the job in a FIFO queue at a leaf within its station. This method involves computing the job priority to decide which queue or queue set should the job be placed in. At the leaf nodes, the MSIBasicQ::Enter(MSIJob *iJob) method actually enters the job at the tail end of the FIFO queue. As shown below, at every intermediate node, each of its subqueues is associated with a range of priorities. The priority computed at intermediate nodes indicates subqueue to enter the job into.

1.2 CALCULATION OF JOB PRIORITY

The calculation of job priority at a QSet (MSIHierarchicalQ) node can be configured differently for different nodes to achieve different objectives in job prioritization. An example would be to split jobs at RootQ based on project and at the next level split jobs based on two user groups. With this configuration, an administrator can allocate resources such as server threads and database connections for each project and user group independently, and specify their servicing schemes as desired. This configuration can be achieved by specifying a priority function at RootQ, which ignores every priority variable other than project and returns the project id as the job priority, which will be used to index into the Qset and enter the job into the corresponding node below RootQ.

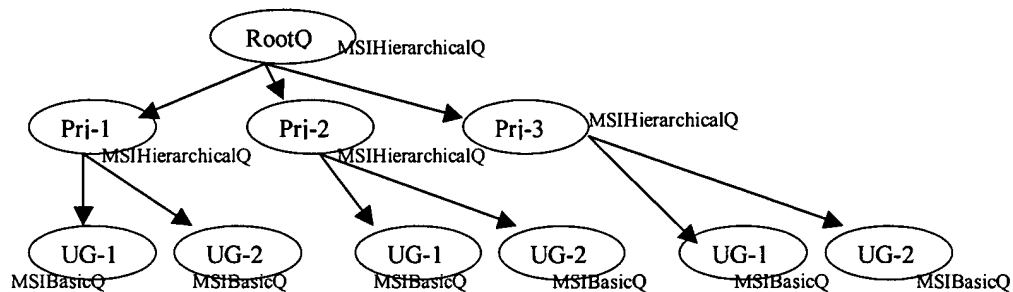


Figure 2. Configuring queue structure so that at the first level, the job is prioritized based on the project and at the second level it is prioritized based on the user group. Note that at this stage, the queue structure represents only a means of categorization. Priority ranges associated with the queues together with servicing policies determine the real priorities between queues and the order in which jobs are processed in a PU.

IMPLEMENTATION OF JOB PRIORITY

1.3 MSIJOBPRIORITYSCHEMEOBJECT

To implement different priority calculations at different nodes of the hierarchical queue, every MSIHierarchical node object contains a MSIJobPrioritySchemeObject, which provides the CalculatePriority method. MSIJobPriorityScheme is an abstract base class which only provides the CalculatePriority interface method. In DSS Server, there are several classes derived from MSIJobPrioritySchemeObject, which implement the calculate priority in various ways. There are several predefined classes derived from MSIJobPrioritySchemeObject defined in the MSIPU.h as described below. For example MSIJobPrioritySchemeObjectRandom is equivalent to entering the job into one of the queue set or queue selected randomly.

```

Class ConfigManager; // forward declaration

template <class JOB_TYPE> class MSIJobPrioritySchemeObject {
protected:
    int mType;
public:
    typedef int PCFunc(JOB_TYPE *iJob);
    MSIJobPrioritySchemeObject(int iType = -1) :mType(iType) {};
    virtual int GetType() { return mType; }
    virtual ~MSIJobPrioritySchemeObject() {};
    virtual int CalculatePriority(JOB_TYPE *iJob) = 0;
    virtual bool Init(ConfigManager *iConfig, StringList & iPath) = 0;
    virtual MSIJobPrioritySchemeObject<JOB_TYPE> *Clone() = 0;
};

enum MSIJobPriorityScheme {
    gcJobPrioritySchemeDefault = 0,
    gcJobPrioritySchemeUserSupplied = 0,
    gcJobPrioritySchemeRandom,
    gcJobPrioritySchemeObjectMapBased
};

class MSIJobPrioritySchemeObjectRandom: public MSIJobPrioritySchemeObject<MSIJob> {
public:
    MSIJobPrioritySchemeObjectRandom():
        MSIJobPrioritySchemeObject<MSIJob>(gcJobPrioritySchemeRandom) {};
    int CalculatePriority(MSIJob *iJob){
        return rand();
    }
    MSIJobPrioritySchemeObject<MSIJob> *Clone() {
        return new MSIJobPrioritySchemeObjectRandom();
    };
    bool Init(ConfigManager *iConfig, StringList & iPath){
        return true;
    }
};

class MSIJobPrioritySchemeObjectUserSupplied: public MSIJobPrioritySchemeObject<MSIJob> {
public:
    MSIJobPrioritySchemeObjectUserSupplied():
        MSIJobPrioritySchemeObject<MSIJob>(gcJobPrioritySchemeUserSupplied) {};
    int CalculatePriority(MSIJob *iJob){
        return iJob->GetPriority();
    }
    MSIJobPrioritySchemeObject<MSIJob> *Clone() {
        return new MSIJobPrioritySchemeObjectUserSupplied();
    };
    bool Init(ConfigManager *iConfig, StringList & iPath){
        return true;
    }
};

```

1.4 CREATION OF MSIJOBPRIORITYSCHEMEOBJECTS

Note that the MSIJobPrioritySchemeObject interface also includes two more methods, namely Init and Clone methods. The Init() method is used to initialize the object with a configuration settings specific to that node. For example, this would be used to which path the object is attached to, so that any other configuration information can be obtained from the ConfigManager. The Clone() method is used by MSIPU in creation of these objects at the MSIHierarchicalQ nodes. MSIPU is given an array of predefined prototype objects, one of each type, from which MSIPU clones all the other objects as the Q structure is being created.

At the DSS Server startup time, based on the configuration specified, config manager creates the processing units with the required queue structure along with the MSIJobPrioritySchemeObjects at all MSIHierarchicalQ nodes. When there is no scheme specified, either the parent node's scheme is used or a default is selected as shown below.

```
// extracted from MSIPU.cpp
MSIPU::BuildHierarchicalQ(...) {
    ...
    // check if PriorityScheme parameter is specified at this node,
    // and if so store its type in IPriorityScheme
    if(!PriorityFunctionKeyFound) {
        // no PriorityFunction key found
        IPrioritySchemeObject = mPrioritySchemeObject[IPriorityScheme]->Clone();
        IPrioritySchemeObject->Init(lSubParameterTree);
    }
    else {
        // no PriorityFunction key here but this is a QSet
        if(!ParentPrioritySchemeObject) {
            // parent got hold of one, clone it
            IPrioritySchemeObject = iParentPrioritySchemeObject->Clone();
            IPrioritySchemeObject->Init(lSubParameterTree);
        }
        else {
            // parent had none: probably this is the root and has no PriorityScheme specified
            IPrioritySchemeObject = mPrioritySchemeObject[gc.JobPrioritySchemeDefault]->Clone();
            IPrioritySchemeObject->Init(lSubParameterTree);
        }
    }
    ...
}
```

1.5 PROPOSED IMPLEMENTATION ACCORDING TO SPECIFICATIONS

The above design of a queue structure with MSIJobPrioritySchemeObject at each node is general enough to implement vastly different ways of determining priorities of a job by way of overriding MSIJobPrioritySchemeObject::CalculatePriority() method in derived classes of MSIJobPrioritySchemeObject. It even allows for different ways of computing priority at different nodes, even though it is not desirable in general. DSS Server specification document fixes many of the dimensions for the sake of simplicity and manageability.

Here is a summary of the specification regarding job priorities. Job priorities are calculated based on a number of factors relevant to the job. Some of the factors are allowed to interact nonlinearly in determining the priority while others are assumed to interact linearly. DSS Server allows for the administrator to select which factors interact nonlinearly and which ones interact linearly. This is selected through a configuration wizard such as the one included in DSS Server Administrator product. Examples of factors that could interact nonlinearly are: project, user group, report type, initial report priority, time period and report cost as determined by a linear combination of other factors. The factors that are assumed to interact linearly are those that can be measured quantitatively for any job. Examples of linear factors are: historical report cost, estimated report cost, number of database queries, size of result set etc. Each linear factor is associated with a weight, and a combined report cost is computed as a weighted linear

combination of all the linear factors. This combined report cost is one of the factors allowed to interact nonlinearly in determining the job priority. The nonlinear factors determine the the job priority through a priority map (or table) whose input dimensions are all the factors allowed to interact nonlinearly and the entry in the map giving the job priority. The map representation to determine job priority is chosen because it is not only powerful enough to represent any kind of relationship between the input dimensions and the resulting priority, but also is convenient to define, store and communicate and efficient to calculate the priorities.

Thus to compute the job priority, first the values of linear factors are computed for the job, their weighted linear combination is computed as a combined report cost. Then, the priority map is looked up using the combined report cost together with all other nonlinear factors as input dimensions into the map. The map entry represents the priority for that job.

Note that the priority map is allowed to be different at different nodes of a process unit and different across process units, allowing for highly flexible priority configurations. In the current design, the nodes contain a pointer to `MSIJobPrioritySchemeObjectMapBased` object, which store a pointer to the priority map for that node. This allows for sharing the priority maps between nodes.

Also, the weights in the weighted linear combination formula are allowed to be different at different PU's. This is done by storing the multiple lists of linear factor name along with its weight in a `DSSProperty` interface in the report instance itself. Each list corresponds to a weighted linear combination formula. These lists are created in the report instance by looking up the report definition, server definition and project definition the former parameters overriding the latter in case of multiple definitions containing these parameters.

To define the priority calculation function as described above, one would specify the `PriorityScheme` parameter to the node as `gcJobPrioritySchemeObjectMapBased`. The `MSIJobPrioritySchemeObjectMapBased` class is as defined below. Individual nodes would get the map information from `ConfigManager`. `ConfigManager` would have access to all the configuration information, including the priority maps defined by the administrator at different nodes. Note that it allows the map to be shared across processing units and nodes as configured, without having to duplicate them.

```
// MSIConfig.h

class MSIJobPrioritySchemeObjectMapBased: public MSIJobPrioritySchemeObject<MSIJob> {
protected:
    ConfigManager *mConfig;    // Config has all configuration info
    StringList mPath;          // path to the SubQ including the name of the PU.
    Int mNonlinearFactors;
    Int *mNonlinearFactor;
    Int *mNonlinearFactorMaxValue;
public:
    MSIJobPrioritySchemeObjectMapBased():
        MSIJobPrioritySchemeObject<MSIJob>( gcJobPrioritySchemeObjectMapBased {});
    int CalculatePriority(MSIJob *iJob){
        void *aPriorityMapPtr = mConfig->GetPriorityMap(mPath);
        int aValue;
        int aNonlinearFactorMaxValue = 1;
        for(int aNonlinearFactor = mNonlinearFactors-1; aNonlinearFactor >=0; aNonlinearFactor--){
            // the combined report cost is one of the nonlinear factors
            // all nonlinear factors are computed within MSIJob module
            aValue = iJob->GetFactorValue(mNonlinearFactor[aNonlinearFactor]);
            aPriorityMapPtr = (void *) ((int *)aPriorityMapPtr + aNonlinearFactorMaxValue*aValue);
            aNonlinearFactorMaxValue *= mNonlinearFactorMaxValue[aNonlinearFactor];
        }
        return *((int *)aPriorityMapPtr);
    }
    MSIJobPrioritySchemeObject<MSIJob> *Clone() {
        return new MSIJobPrioritySchemeObjectMapBased ();
    }
};
```

```
};  
bool Init(ConfigManager *iConfig, StringList & iPath){  
    mConfig = iConfig;  
    StringListCopy(mPath, iPath);  
    return true;  
}  
};
```

1.6 CHANGING PRIORITY AFTER JOB IS ENTERED INTO QUEUE

Once the job is entered into a queue, its priority is no longer relevant, as the servicing schemes dictate in which order jobs are serviced within a queue (servicing schemes are described in the next section). Hence, changing its priority would not affect the order in which it is serviced. If an administrator wishes to process the job earlier or later than it would have otherwise (which is usually thought of as changing the priority of a job), DSS Server allows for several commands which allow flexible job movement within or across queues.

The DSS Server commands that allow moving a job within a basic queue are:

- MoveAheadByOne: moves job ahead by one (no effect if there is no other job in front of this one)
- MoveBehindByOne: moves job behind by one (no effect if there is no job behind this one in the queue)
- MoveToFront: moves the job to front of the queue
- MoveToBack: moves the job to the back of the queue

For the above commands, the queue need not be supplied by the client, as DSS Server will find the queue in which the job is waiting and carry out the command within that queue.

The DSS Server commands that allow moving a job across queue are:

- MoveToQueue: given a job and the new queue name (which could be a hierarchical queue), DSS Server will find the queue in which the job currently is waiting, remove from there and enter it into the new queue specified.

JOB SERVICING SCHEMES

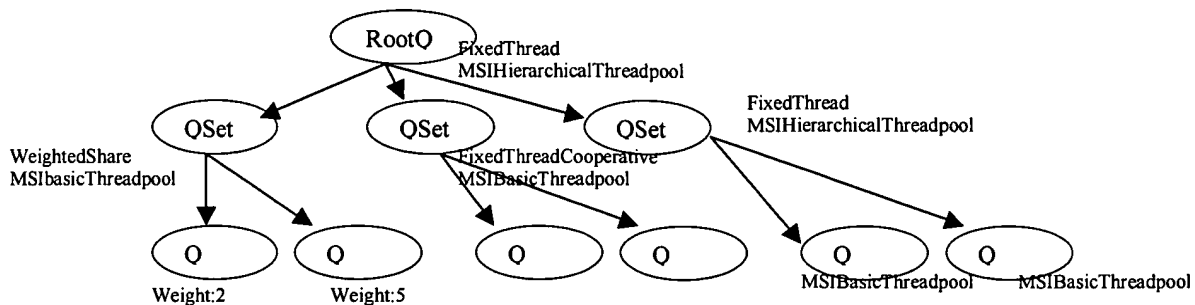
Job servicing schemes are mechanisms to control how jobs are serviced. So, the servicing scheme is applied when these resources are allocated to the jobs for their processing. That is, whenever all resources (such as server thread and database connections) become available to process a job, which job would they be allocated to next. Notice that the threads are the primary resources in a process unit, which does not do tasks involving heavy database usage. One exception, is the case of a database process unit that executes warehouse queries, where database connections (which are paired with the thread that created them) are the primary resource.

Given the fact that jobs are always serviced first-in-first-out within a queue, the job-servicing scheme in effect applies to selection of queues. That is, any time a resource is available to process a job, it is sufficient to select a queue to service next. Thus, servicing schemes are attached to a Qset, where there is a choice of queues to select from (whenever we decide to process jobs from that Qset). It follows that every Qset node in the process unit hierarchy, contains a servicing scheme specified to it. Accordingly, every MSIHierarchicalQ has a mSubQServicingScheme data member, of integer type that specifies the servicing scheme from an enumeration of available servicing schemes. The available choices of servicing schemes defined below.

```
enum MSISubQServicingScheme {
    gcSubQServicingSchemeUndefined = -1,
    gcSubQServicingSchemeDefault = 0,
    gcSubQServicingSchemeFixedThreadCooperative = 0,
    gcSubQServicingSchemeFixedThread,
    gcSubQServicingSchemeHighestPriorityFirst,
    gcSubQServicingSchemeWeightedShare
};
```

1.7 UNITS OF INDEPENDENT RESOURCE ALLOCATION AND CONTROL

In a process unit, the unit at which resources can be independently allocated and controlled are those Queues and Qsets that have fixed servicing schemes. A Qset that specifies a servicing scheme other than fixed implies that the subqueues within it are serviced collectively. This also implies that resources should be allocated collectively. Thus, the current design allocates independent thread pools to all Qsets that have servicing schemes other than fixed. At the Qset nodes that specify a fixed servicing scheme, the subqueues within it are serviced independently on their own, and thus have their own collection of thread pools. In this case, we associate the pool of subqueue thread pools to the Qset, to conveniently add or remove threads from the Qset. As an example, we show the servicing schemes at each node and the resulting thread pool hierarchy. MSIBasicThread pools is a collection of MSIThreads. MSIHierarchicalThreadPool is a collection of MSIBasicThreadpools.



1.8 JOB PROCESSING ACCORDING TO A SERVICING SCHEME

In our process unit design, the job processing in a process unit takes place via the MSIQTask class. MSIQTask::Run() method is an encapsulation of the task to be performed within the process unit whenever the processing resources are available. Typically, the server threads are the resources, which come from a hierarchical thread pool within the process unit. The threads within a thread pool are given an object of MSIQTask, with the MSIBasicQ node or an MSIHierarchicalQ node which they service. The method GetNextJob implements checking all subqs from a given level for a job. It returns timeout if there is no job after a scan of all basic q's under the given level.

```
void MSIQTask::Run(){
    ...
    // Code deleted: setup preferred path if any and
    // setup current weights for all subqueues with weighted share
    ...
    while(!IsCanceled()){
        if(mPreferredSubQ){
            IRC = GetNextPreferredJob(mPreferredSubQ, 0,
                                     mSubQIndexBelowFixed.begin(), &IJob, &IQ);
            IRC = mPreferredQ->WaitForNext(&IJob, gcMSIQMaxWaitTimeForNextJob);
            if(IRC==gcMSIQTimedOut){
                continue;
            }
            IRC = GetNextJob(mQ, 0, mFixedSubQIndex.begin(),
                            mNonPreferredSubQIndex.begin(), true, &IJob, &IQ);
            if(IRC == gcMSIQTimedOut){
                ITimedOut = true;
                continue;
            }
            // else got job in a non preferred Q
        }
        // else got job in preferred Q
    }
    else{
        // no preferred Q
        IRC = GetNextJob(mQ, 0, mFixedSubQIndex.begin(),
                        mNonPreferredSubQIndex.begin(), true, &IJob, &IQ);
        if(IRC == gcMSIQTimedOut){
            ITimedOut = true;
            continue;
        }
    }
    ...

    mCheckGovernorsTask->SetJob(IJob);
    mCheckGovernorsTask->Run();
    if(mCheckGovernorsTask->GetStatus() == MSIJobTask::JOBTASK_ABORT);
    ...
}
else{
    ...
    IJob->AddRef(); // we need the job to be alive till we are done with it
    IJobId = IJob->GetId();
    IJob->SetPUStart(mOwnerId);
    try{
        mProcess->SetJob(IJob);
        mProcess->SetThread(GetThread());
        mProcess->Run();
    }
    catch(...){
        //MSIQTaskTrace(_TEXT("ERROR: task execution for job threw an exception"),
        IJobId, IQ);
    }
    mProcess->SetThread(NULL);
}
```

```

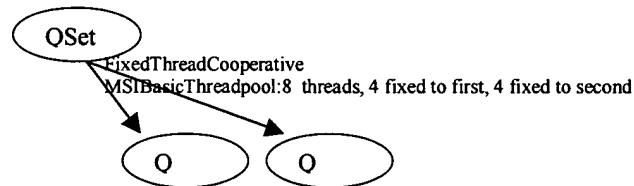
    }
    lJob->SetPUExit(mOwnerId);
    IQStatus = IQ->Dequeue(lJob);
    if(IQStatus == gcMSIQSucceeded) lJob->Release();
    lThreadInfo.SetJobId(-1);
    ...
} // end of while
...
}

```

1.8.1 FixedThreadCooperative

When the servicing scheme is FixedThreadCooperative, the MSIQTask then also contains a PreferredSubQ path, which is the path down the hierarchy along which all nodes specify fixed thread cooperative. In this case, threads attempt to get a job from the preferred q/qset first and if failed, will attempt to get from any queue in the Qset. The method GetNextPreferredJob implements checking the preferred subq path for any job, which returns timeout if there is no job after one scan.

Note: This servicing scheme is meant to fix threads to subqs, but let them service other subq's at the same level only when there is no job in the subq to which the thread is fixed. Assume the following scenario.



A Qset contains two subqs, which are basic queues. The number of threads in the threadpool are eight, out of which four are fixed to first and the remaining four are fixed to the second. Suppose, at a particular instance the first subq had no jobs, and the four threads fixed to started processing jobs from the second subq which had a large number of jobs. Suppose a job arrives in the first subqueue during which time all four threads are still servicing the jobs they picked up from the second subq. At just such a time, one of the threads fixed to the second subq finishes one of the jobs and becomes available. Which subq will it pick the next job from?

There are two different interpretations of this servicing scheme which answer this question differently. The first interpretation is that the threads are fixed to the preferred subq they are servicing and as long as there are jobs in that subq, they do not attempt to take jobs from any other queue. According to this, the answer to the above question would be that the thread belonging to the second subq would take its next job from the second subq.

The second interpretation of this servicing scheme is that the threads are not fixed to the subq permanently, rather they are dynamically assigned to a subq's according to a fixed proportion given in the configuration. According to the above thread (any thread from the second subq that becomes available next), gets assigned to the first subq as soon as there is a job in the first subq. So, the threads would change their preferred queue to optimize in the situations described above. This makes more sense as automatic load balancing within a process unit.

In our current implementation, we have chosen to use the first interpretation and minimize the implementation complexity.

1.8.2 WeightedShare

When the servicing scheme is weighted share, the subqueues within the Qset also incorporate weights and a running count of number of jobs serviced at each subqueue. The method `GetNextJob` checks all subqs at the given level or a job and implements the running weight counter increments in a queue when returning a job from a that queue. The running counts are reset to zero when last subq count at any level reaches its maximum value.

1.8.3 HighestPriorityFirst

The `GetNextJob` method implements checking all subq's at a given level or a job, in the order of highest priority to lowest priority subq.